Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016)
Sofia, Bulgaria

Jesus Carretero, Javier Garcia Blas, Svetozar Margenov
(Editors)

October, 6-7, 2016

Schlatter, P., & Peplinski. A., N. (2016). Highly Tuned Small Matrix Multiplications Applied to Spectral Element Code Nek5000. En *Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016)* Sofia, Bulgaria (pp. 69-72). Madrid: Universidad Carlos III de Madrid. Computer Architecture, Communications, and Systems Group (ARCOS).

Highly Tuned Small Matrix Multiplications Applied to Spectral Element Code Nek5000

Berk Hess, Jing Gong, Szilárd Páll

KTH Royal Institute of Technology, Sweden
hess,gongjing,pszilard@kth.se

Philipp Schlatter, Adam Peplinski

Department of Mechanics, KTH Royal Institute of Technology, Sweden
pschlatt,adam@mech.kth.se

Abstract

Nek5000 is an open-source code for simulating incompressible flows using MPI for parallel communication. In the Nek5000 code, the tensor-product-based operator evaluation can be implemented as small dense matrix-matrix multiplications. It is clear that the routines for calculating the matrix-matrix product dominate the execution time of Nek5000. In this paper, we conduct the optimization of matrix-matrix multiplication using SIMD intrinsics and the LIBXSMM package. The evaluation of the computational cost and optimization of these subroutines is not only applied to the CFD code Nek5000, but also to the NekCEM and NekLEM software, which share same data structures with Nek5000.

Keywords Spectral Element Method (SEM), Nek5000, Nekbone, Single instruction multiple data (SIMD), LIBXSMM

I. Introduction

Nek5000 [1] is an open-source code for simulating incompressible flows using MPI for parallel communication. The code is widely used in a broad range of applications. The Nek5000 discretization scheme is based on the spectral-element method [2]. In this approach, the incompressible Navier-Stokes equations are discretized in space by using high-order weighted residual techniques employing tensor-product polynomial bases. The tensor-product-based operator evaluation can be implemented as small matrix-matrix multiplication, The main part of the program Nek5000 consists in small matrix-matrix multiplication routines, in which the program spends most of its time (more than 60% in a 2D version) [3].

Currently, the routines are basic FORTRAN routines with nested loops to compute the matrix multiplications in Nek5000. The aim of the work is to enhance the routines using vectorization techniques like SIMD (Single Instruction Multiple Data) instructions [4] and the high performance library for small matrix multiplications LIBXSMM [5].

The remainder of this paper is organized as follow. Section 2 describes the algorithms and the SIMD implementations. Section 3 presents the main performance results. Finally the conclusions and further works are discussed in Section 4.

II. The algorithms and the SIMD implementation

In Nek5000, the small dense matrix multiplication is written as

$$\mathbf{C}_{n_1 \times n_3} = \mathbf{A}_{n_1 \times n_2} \mathbf{B}_{n_2 \times n_3}$$

where the size of n_1, n_2, and n_3 can be N or N^2 with typical $N \in (4-16)$. In the routine written as below we use the "C" ordering wherein columns of **B** are assumed stored consecutively and that successive rows of **A** are stored n_1 floating point words apart in memory, see [7]).

```
int i, j, k;
for (i = 0; i < n1; i++) {
   for (k = 0; k < n3; k++) {
      c[i][k] = 0.0;
      for (j = 0; j < n2; j++) {
```

```
          c[i][k] += a[i][j] * b[j][k];
      }
    }
  }
```

However this implementation is very time consuming since the compilers have a hard time optimizing and vectorizing it. Also there is no hint given for the values of the loop parameters, and the compiler would not take full advantage of the underlying SIMD architecture.

The principle of SIMD instruction is to apply an instruction to multiple operands at once instead of on one operand and thus considerable improving code performance. Recent processors, e.g. Intel Haswell, have support for 256-bit SIMD instructions that operate on 256-bit registers [6] (512-bit for the next generation), thus processing 4 double precision numbers simultaneously. With 2 fused multiply-add operations per cycle per core, this results in a peak throughput of 16 FLOPs per cycle per core. However, with standard code one has to rely on the compiler to extract sufficient SIMD vectorization. Except for triial cases, such as operations on large vectors, this is a difficult task. Furthermore, the throughput is often limited by speed with which the operands can be loaded from memory or L2/L3 cache into SIMD registers.

```
int i, j, k;
for(k = 0; k + 1 < n3; k += 1) {
   simd_db bs0[n2];
   for (int j = 0; j < n2; j++) {
       bs0[j] = simd_broadcast_sd(b + j + k * n2);
   }

   i = 0;
   while(i + SIMD_WIDTH <= n1) {
       simd_db as = simd_loadu_pd(a + i);
       simd_db c0 = simd_mul_pd(as, *bs0);

       for (int j = 1; j < n2; j++) {
           as = simd_loadu_pd(a + i + j*n1);
           c0 = simd_fmadd_pd(as, bs0[j], c0);
       }
        simd_storeu_pd(c + i + k * n1, c0);
        i += SIMD_WIDTH;
   }
   if (i < n1) {
     simd_si mm = simd_castpd_si(simd_loadu_pd(
                  (const double*)mask[n1-i]));
     simd_db as = simd_maskload_pd(a + i, mm);
     simd_db c0 = simd_mul_pd(as, *bs0);
     for (j = 1; j < n2; j++) {
        as = simd_maskload_pd(a + i + j*n1, mm);
        c0 = simd_fmadd_pd(as, bs0[j], c0);
     }
     simd_maskstore_pd(c + i + k * n1, mm, c0);
   }
}
```

To optimize the matrix-matrix multiplication routines in the program we firstly take maximum advantage of the underlying architecture by using SIMD intrinsics, supported by several compilers, and to help the compiler by unrolling the different loops that are involved in the routine [8]. The fact that we have stride-1 access within the j-loops and not necessarily within the i-loops at the same time makes this idea less appealing. Thus, we could aim at SIMD vectorizing to ensure that all j-loops will SIMD vectorize well in the matrix-matrix multiplication and this requires that the compiler does indeed recognize the j-loops as stride-1 loop. By using the instruction set provided for AVX2-compatible architecture (`_mm256_*`). One instruction has been replicated 4 times in $\mathbf{B}$ and 4 rows in $\mathbf{A}$ are computed simultaneously.

$$\begin{bmatrix} |\mathbf{B}_{1,1}| & \cdots & \mathbf{B}_{1,p} \\ \mathbf{B}_{2,1} & \cdots & \mathbf{B}_{2,p} \\ \mathbf{B}_{3,1} & \cdots & \mathbf{B}_{3,p} \\ \mathbf{B}_{4,1} & \cdots & \mathbf{B}_{4,p} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{n,1} & \cdots & \mathbf{B}_{p,p} \end{bmatrix} \quad \begin{bmatrix} |\mathbf{A}_{1,1}| & \cdots & \mathbf{A}_{1,n} \\ |\mathbf{A}_{2,1}| & \cdots & \mathbf{A}_{2,n} \\ |\mathbf{A}_{3,1}| & \cdots & \mathbf{A}_{3,n} \\ |\mathbf{A}_{4,1}| & \cdots & \mathbf{A}_{4,n} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{n,1} & \cdots & \mathbf{A}_{n,n} \end{bmatrix}$$

Using assembly code can further optimize loops and memory fetching wherever possible and manually unroll. In Algorithm 1 we shown the core of assembly code for the routine. Most of the loops go downwards instead of the natural upward scheme. We keep in register everything that is often need (loop indexes) to avoid redundancy when possible.

The Intel LIBXSMM is designed in a very flexible way, that is, separated into a frontend (routine selection) and backend (specific xGEMM code generation). As a result, LIBXSMM can achieve its high application level performance for Intel processors. LIBXSMM offers an auto dispatcher which decides which backend should be executed for the given parameter set [5]. Finally we call through the interface of LIBXSMM, which implements the matrix multiplication shown in Algorithm 2 [9].

III. Performance results

To understand the performance implications of SIMD optimization, this paper presents case studies of porting and optimization of kernel benchmarks for a spectral element code Nekbone, which is a simplified version of a computational

Algorithm 1 Assembly SIMD code

```
for_j_loop:
        subq    %r9, %r14
        load_bs0_array

        #Initialisation of i-loop
        movl    %r8d, %r11d
        subq    %r8, %rdi
        subl    $32, %r11d
        jle     while_i_loop_end
while_i_loop:
        loop_mult       for_k_loop
        subl            $32, %r11d
        jg              while_i_loop
while_i_loop_end:
        loop_mult       for_k_loop_in_n1, 1
        decl            %r10d
        jge             for_j_loop
end_of_function:
        popq    %r15
        popq    %r14
```

Algorithm 2 LIBXSMM Interface

```
CALL libxsmm_init()
CALL libxsmm_dispatch(xmm,            &
             n1, n3, n2, alpha=alpha, beta=beta)
IF (libxsmm_available(xmm)) then
   CALL libxsmm_call(xmm, C_LOC(ap),  &
             C_LOC(bp), C_LOC(cp))
ENDIF
CALL libxsmm_finalize()
```

fluid dynamics (CFD) code Nek5000. Nekbone focuses on the Poisson operator evaluation that is a central computational kernel in Nek5000. As kernel benchmarks, we focus on highly tuned matrix multiplications for fine-grained parallelism of matrix-vector multiplications.

An initial performance profiling of Nek5000 application on a single Haswell node was carried out using the Cray Performance Analysis Tools (CrayPAT) profiler. The goal of this profiling work was to identify which subroutines are the most time consuming and can provide enough workload to exploit the SIMD instructions. The profiling table above shows the profiling results. The subroutine `mx2mf2` for the matrix multiplication takes around 42.3% total executive time.

Table 1: Profile by Function

Samp%	Samp	Group / Function
100.0%	2811.0	Total
95.7%	2689.0	USER
42.3%	1190.0	mxmf2_
14.5%	408.0	cg_
12.3%	347.0	glsc3_
11.3%	319.0	add2s2_
4.0%	113.0	add2s1_
3.1%	86.0	jl_gs_gather
2.0%	55.0	jl_gs_scatter
1.7%	47.0	add2_
1.7%	47.0	jl_sortp_ull
1.3%	37.0	jl_sortp_ui
4.3%	120.0	ETC

We carry out the performance tests on Beskow which is a Cray XC40 system, based on Intel Haswell processors and Cray Aries interconnect technology. This system has Intel Xeon E5-2698v3 (Haswell) CPUs with processor frequency of 2.3 GHz.

Figures 1 and 2 show the performance results with number of elements $E = 10000$ and $E = 20000$, respectively. From these figures, we find that better performance can be obtained using the SIMD intrinsics and LIBXSMM. Also SIMD intrinsic code can lead high performance with lower orders of polynomial ($N = 4, 6, 8$).

IV. Conclusion and Future Work

We have studied the performance implications of several optimization of small matrix-matrix multiplication. Specifically an originally optimized version is adapted to Nek5000. Through the SIMD vector instructions and the Intel library LIBXSMM, the results show that the performance significantly improved on the matrix multiplication. The overall performance of Nek5000 has also been improved due to the use of SIMD instructions.

References

[1] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, Nek5000 web page, Web page: http://nek5000.mcs.anl.gov.

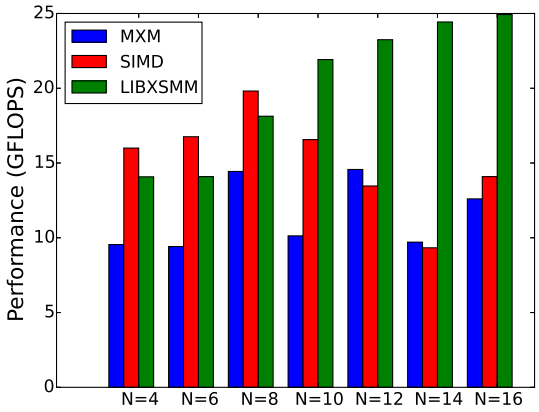

Figure 1: Performance results on a Haswell node with number of elements $E = 10000$

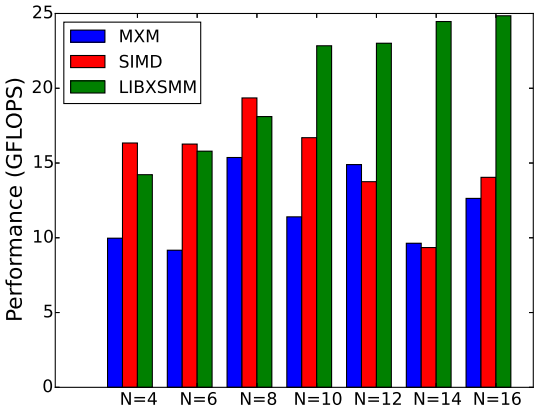

Figure 2: Performance results on a Haswell node with number of elements $E = 20000$

[2] M. Deville, P. Fischer, and E. Mund, *High-order methods for incompressible fluid flow*, Cambridge University Press, 2002.

[3] P. Fischer, J. Lottes, W. D. Pointer, and A. Siegel, "Petascale Algorithms for Reactor Hydrodynamics", *Journal of Physics: Conference Series*, vol. 125, 012076, 2008.

[4] Intel Architecture Instruction Set Extensions Programming Reference, www.naic.edu/ phil/software/intel/319433-014.pdf

[5] A. Heinecke, H. Pabst and G. Henry, "LIBXSMM: A High Performance Library for Small Matrix Multiplications," in *the Proceedings of SC15*, Austin, USA, November 15-20, 2015.

[6] G. Mitra, B. Johnston, A.P. Rendell, E. Mccreath, and J. Zhou, "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. in *IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*, pp 1107-1116, 2013.

[7] W. P. Petersen and P. Arbenz, *Introduction to Parallel Computing, A Practical Guide with Examples in C*, Oxford University Press, 2004.

[8] S. Páll and B. Hess "A flexible algorithm for calculating pair interactions on SIMD architectures", *Computer Physics Communications*, vol. 184, no. 12, pp. 2641–2650, 2013.

[9] M. Hutchinson, A. Heinecke, H. Pabst, G. Henry, M. Parsani and D. Keyes, "Efficiency of High Order Spectral Element Methods on Petascale Architectures," in *ISC High Performance 2016 LNCS 9697, J.M. Kunkel et al. (Eds)*, pp. 449–466, 2016.

Acknowledgment

This work is partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS) and the Swedish e-Science Research Center (SeRC).

We would also like to thank Erik Lindahl and Ismael Bouya for help with the SIMD code as well as Alexander Heinecke, Hans Pabst, and Greg Henry from Intel for the LIBXSMM used in the paper.